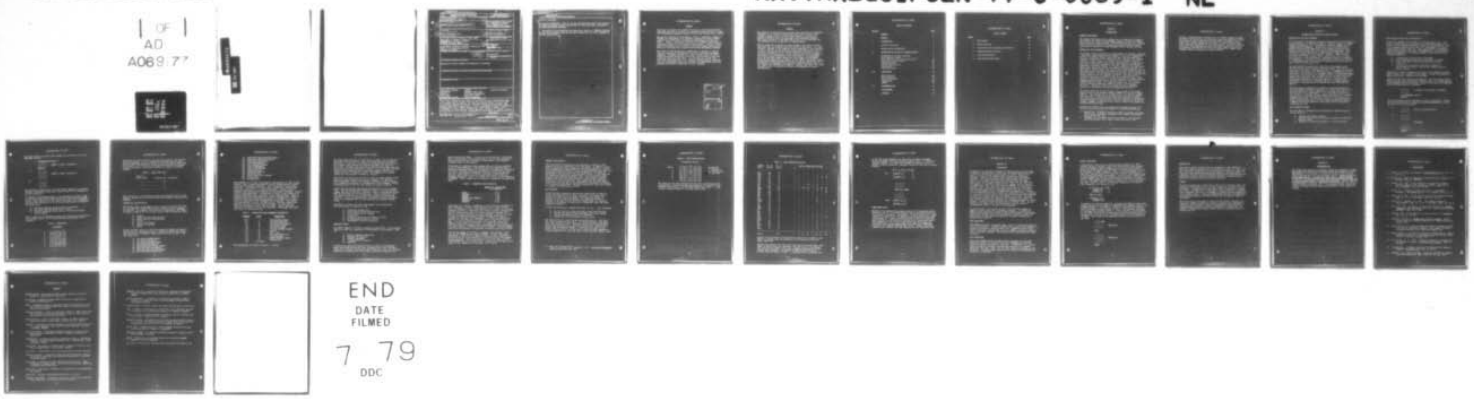
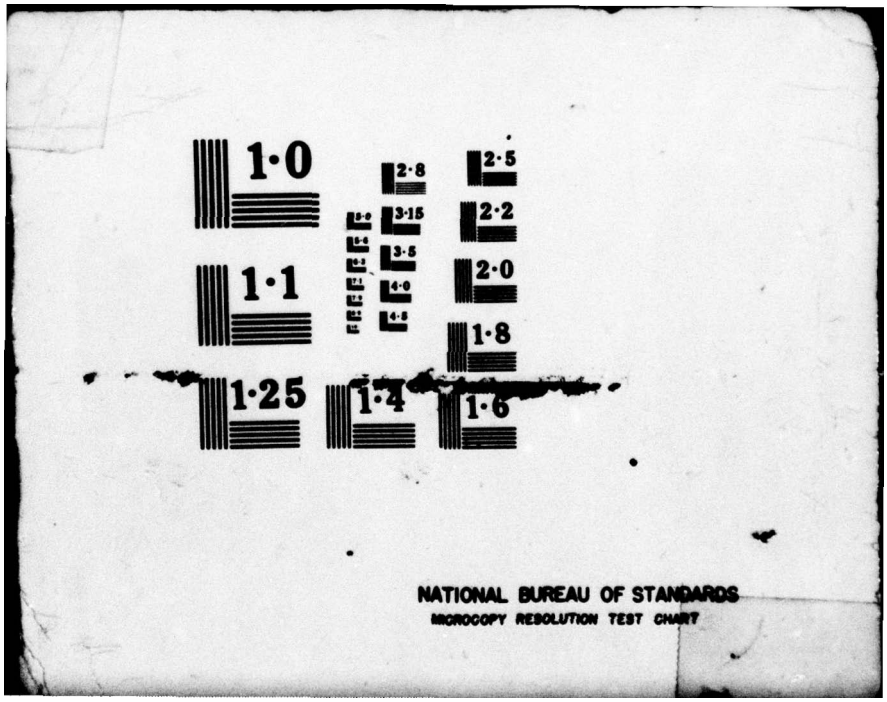
AD-A069 177 UNIVERSITY OF CENTRAL FLORIDA ORLANDO DEPT OF COMPUT--ETC F/G 9/2
LANGUAGE DESIGN USING DECOMPILATION.(U)
MAY 79 D A WORKMAN

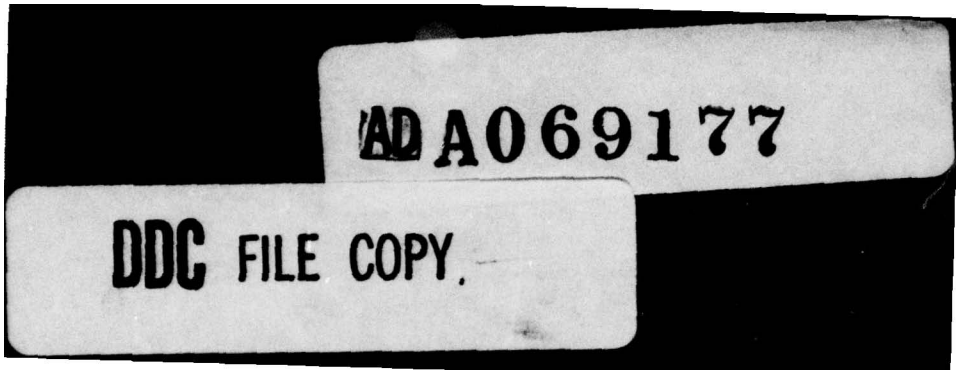
UNCLASSIFIED

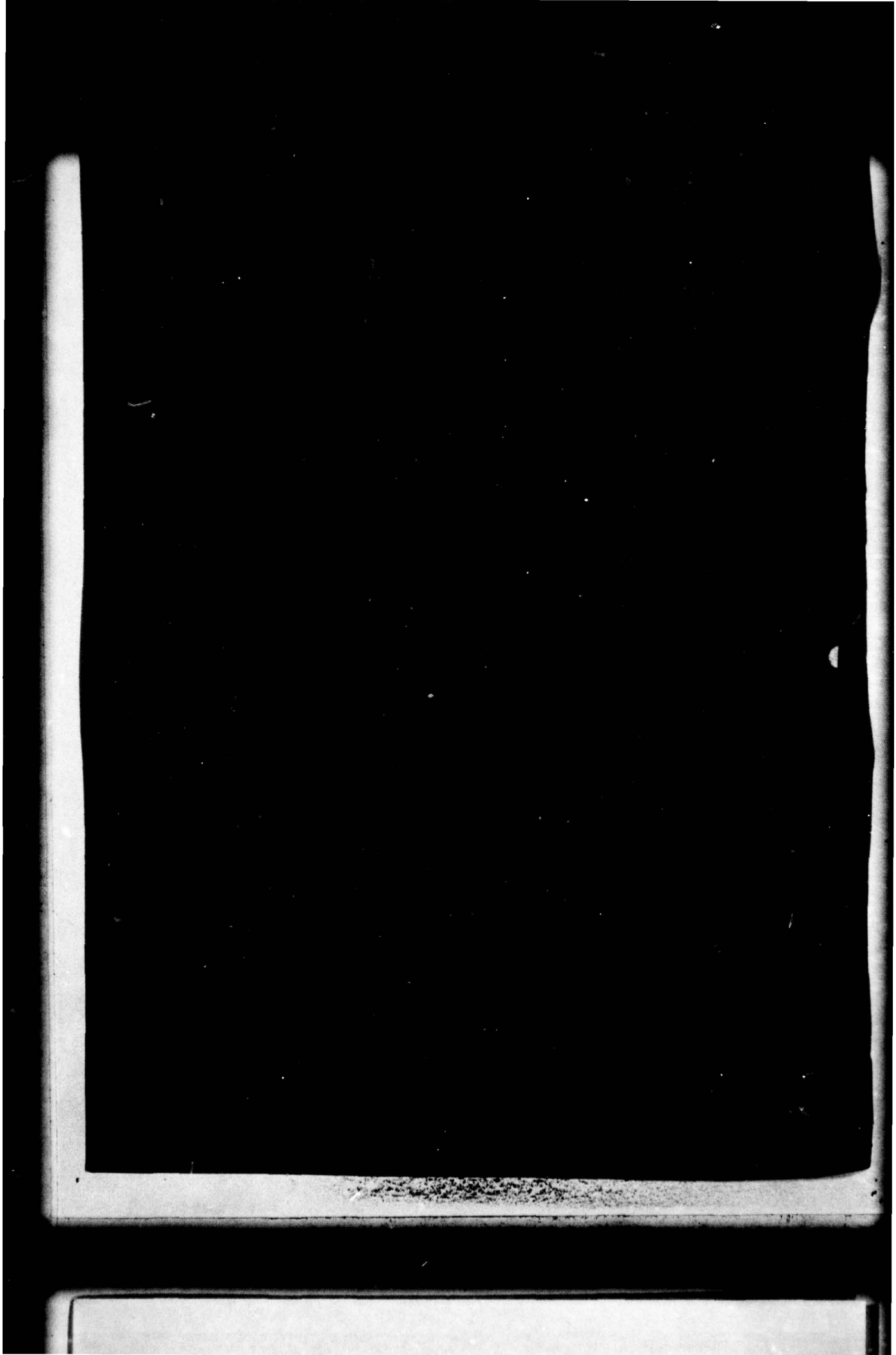
N61339-77-C-0069
NAVTRAEQUIPCEN-77-C-0069-1 NL



END
DATE FILMED
7 79
DDC


1·0
1·1
1·25
2·8
3·15
3·5
4·0
4·5
1·4
2·5
2·2
2·0
1·8
1·6
NATIONAL BUREAU OF STANDARDS
MICROCOPY RESOLUTION TEST CHART


AD A069177
DDC FILE COPY



UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER: NAVTRAEQUIPCEN 77-C-0069-1	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle): LANGUAGE DESIGN USING DECOMPILATION.	5. TYPE OF REPORT & PERIOD COVERED: Final Report for period Aug 77 through Dec 78
6. PERFORMING ORG. REPORT NUMBER	
7. AUTHOR(s): David A. Workman Ph.D.	8. CONTRACT OR GRANT NUMBER(s): N61339-77-C-0069
9. PERFORMING ORGANIZATION NAME AND ADDRESS: University of Central Florida, Orlando, Department of Computer Science, Orlando, Florida 32816	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: NAVTRAEQUIPCEN Task No. 5741
11. CONTROLLING OFFICE NAME AND ADDRESS: Experimental Computer Simulation Laboratory, N74, Naval Training Equipment Center, Orlando, FL 32813	12. REPORT DATE: May 1979
	13. NUMBER OF PAGES: 24
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)	15. SECURITY CLASS. (of this report): Unclassified
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for Public Release; Distribution is unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

High-Level Language	FORTRAN	Control Structures
Decompilers	Assembly level code	
Compilers	Structured programming	
Data types	Real-time systems	
Instruction classes	Control sequences	

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This report represents the results of a project in which decompilation techniques were used to identify the essential characteristics of a high-level programming language suitable for real-time training device systems. The project consisted of three phases. First, a decompiler written in FORTRAN was implemented to map assembly language for a Xerox SIGMA 7 computer into a collection of tables and data forming the basis for phase 3. Second, a structures collection of 33 modules representing an operational system for an F-4

(Continued)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE S/N 0102-014-6601 |

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

411200

Item 20. Continued

trainer was decompiled. Finally, the data gathered from phase 2 was analyzed to identify language features appropriate for some high-level, application-oriented programming language.

The results of the decompilation showed that a hybrid of FORTRAN including bit-strings and locator data was the most appropriate high-level language for trainer applications.

SUMMARY

This report represents the results of a project in which decompilation techniques were used to identify the essential characteristics of a high-level programming language suitable for real-time training device systems.

The project consisted of three phases. First, a decompiler, written in FORTRAN, was implemented to map assembly language for a Xerox SIGMA-7 computer into a collection of tables and data forming the basis for Phase 3. Second, a structured collection of 33 modules representing an operational system for an F4 trainer was decompiled. Finally, the data gathered from Phase 2 was analyzed to identify language features appropriate for some high-level, application-oriented programming language.

The results of the decompilation showed that in addition to features typically supported in conventional scientific-oriented programming languages, a language suitable for trainer systems similar to the F4 should include bit-string and locator data; locator data is data that indirectly references other data. It was found that dynamic data structures and recursion were not essential whereas control mechanisms like the IF-THEN-ELSE, Zahn-loops, DO-WHILE and REPEAT-UNTIL were very prominent and should be supported in such a language.

PREFACE

The objective of this project is to apply known and proven decompilation techniques in an attempt to identify patterns and structures within assembly language source code that might suggest or correspond to familiar constructs found in many high-level programming languages. This effort was novel in the sense that no one, to our knowledge, has used decompilation as an approach to language design.

Several people were involved at various stages of this project and we take space here to recognize those who shared in the effort. Dr. Terry Frederick, Chairman of the Computer Science Department at the University of Central Florida (formerly Florida Technological University), was co-principal investigator on this project and acted primarily as project administrator and consultant. Dr. Ron Dutton, also a faculty member in the Computer Science Department, UCF, contributed to the programming effort and provided support during the early stages of development. Among the most dedicated contributors to this effort were Robert Larsen and Sasidharan Menon (graduate assistants at UCF) who shared most of the programming and debugging effort and spent seemingly endless hours in the computer lab running the decompiler. Finally, Tami Bonar, who joined the project near its completion, helped with debugging and program documentation.

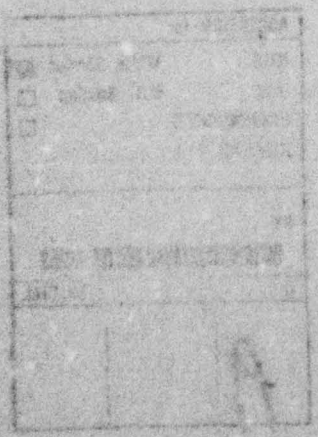


TABLE OF CONTENTS

LIST OF TABLES

SECTION I

INTRODUCTION

CONTRACT OBJECTIVES

The primary objectives of this contract were to identify and formulate the features of a high-level programming language suitable for implementing aircraft trainer systems. The specification for such a language was to be derived from data collected on research systems currently operational at the Naval Training Equipment Center, Orlando, Florida.

METHODOLOGY AND BACKGROUND

Four design criteria determine to a large measure the features comprising a high-level, application-oriented language. First, the language must contain as primitive elements, those data types and data operations intrinsic to problems in the application area. Secondly, the language should be "robust"; that is, it should allow sufficient variety of constructs to make possible the precise representation of computations and control structures frequently occurring in application problems. Robustness is important because it gives the compiler writer more information about the precise nature of a computation making possible the generation of better object code; this is particularly important in real-time applications like those encountered in aircraft trainer systems. A third criterion is the availability of features that add to the self-documenting properties of the language; features of this kind include constructs that induce structure in the control and data flow. This criteria is important because it increases readability, thereby, making debugging and maintenance easier and more effective. Finally, the language should be high in its expressive power; that is, it should provide constructs permitting complex processes to be expressed succinctly. Expressive power in a language increases programmer productivity and tends to diminish his propensity for error.

The primary objective of this contract was to analyze an F4 trainer system written in the assembly language, SYMBOL, for the Xerox SIGMA-7 computer. This analysis was to be carried out in an effort to gather data sufficient to indicate the language constructs best satisfying the design criteria described above. Because a high-level, application-oriented language was desired and because the F4 system was written in assembly language, decompilation was a natural choice as an approach to obtaining the desired data.

Decompilation methods have been studied and developed by Housel[1] and have been used with success by Friedman[2] in transporting operating

1. Housel, B.C. "A Study of Decompiling Machine Languages into High-Level Machine Independent Languages", Tech. Report, CSD-TR-100, Purdue University, 1973.
2. Friedman, F.L. "Decompilation and the Transfer of Mini-Computer Operating Systems", Ph.D. Thesis, Purdue University, 1974.

systems. Although complete decompilation of assembly language programs is normally performed with some predefined high-level syntax as a target, early stages of the decompilation process produce information that is essentially language independent and helpful in indicating what high-level constructs are present in the source program. In the next section, we describe the decompilation process more fully, identify the type of data gathered and show how this data was interpreted to suggest those language features satisfying our design criteria.

SECTION II

DECOMPILATION APPLIED TO LANGUAGE DESIGN

DECOMPILATION AND DATA GATHERING

The first phase of decompilation performs essentially the inverse function of code generation in compilation; that is, symbolic machine or assembly language is mapped "up" to an intermediate low-level language that is largely machine independent. During this process statistics can be gathered concerning the composition of the original source code with regard to special operations, data types and addressing modes. This information is most significant in determining the simple data types that must be supported in the high-level language. Composite data structures like arrays, stacks, queues, linked lists, etc., are much more difficult, if not impossible, to recognize during the first phase of decompilation. Detection of more complex data types is possible only by considering groups of instructions in combination with the control-flow structure of the source code.

The second phase of decompilation identifies code segments called "blocks" and analyzes the control flow among blocks. The effect of this phase is to reduce a program module to a "flow graph" so that high-level control structures (e.g. FOR and DO loops, IF-THEN-ELSE) can be identified. The structure of a flowgraph together with knowledge regarding the functional properties of program blocks can suggest the use of complex data structures. Flow analysis can also identify regions of the program absorbing relatively large amounts of execution time. Information of this kind is extremely important in suggesting what optimization techniques should be employed and where they should be applied.

The third phase of decompilation normally performs code generation from the program's flowgraph representation to high-level object code. Since the high-level source language was an unknown, this phase of decompilation was not implemented. It was the primary objective of this contract to determine the most important features of this unknown language based on the data gathered by the first two phases of the decompilation process. We, therefore, present in the following paragraphs a detailed description of each of these decompilation phases, a summary of the data collected in each phase and our conclusions concerning this data.

BLOCK GENERATION PHASE

The first phase of decompilation was designed to accomplish the following objectives:

1) Identify and classify "blocks";
2) Classify instructions and determine frequency distributions based on class;
3) Determine a frequency distribution of addressing modes.

BLOCK IDENTIFICATION AND CLASSIFICATION

A "block" is defined to be a sequence of instructions, $I_1, I_2, \ldots, I_n$, with I_1 representing the only entry point to the sequence and if for some k, I_k is a transfer instruction (condition or unconditional), then I_j is a transfer instruction for each $j>k$. Furthermore, if I_k is an unconditional transfer, then $k=n$. Each program module was decomposed into blocks during the first phase of decompilation. An instruction, I defined the beginning of a new block whenever I satisfied one of the following conditions:

1) I represented an entry point to the module.
2) I was referenced by some instruction within the module.
3) I was a nontransfer instruction following a transfer instruction.
4) I was the first instruction following a sequence of instructions from which I could not be reached except by a direct transfer.

Condition 4 is really subsumed by the others if the assumption is made that no "dead" code is present in any module. To keep the decompiler as general as possible, condition 4 was included.

Blocks fall into three functional categories. The first category represents all blocks that evaluate some condition prior to a logical decision point dependent upon that condition. Blocks of this type can be modelled by the following high-level statements.

```
_________
_________      A sequence of nontransfer statements
_________

IF (EXPRESSION) GOTO L1
GOTO L2
```

The second category can be identifed as "loop initialization." Blocks in this category contain no transfer instructions themselves but occur just prior to a loop entry point.

```
L1      _________
        _________      Loop Initialization Block
        _________

L2      _________
        _________
        _________
           .
           .           Loop Body
           .
        _________
        _________
        _________
        GOTO L2
```

The third category contains blocks forming the alternatives of an IF-THEN-ELSE construct.

```
          IF(EXPRESSION) GOTO L1
          _____
          _____          BLOCK 1 ("else" alternative)
          _____
          GOTO L2
L1
          _____
          _____
          _____          BLOCK 2 ("then" alternative)
          _____
          _____

L2        _____
          _____
          _____
          _____
```

The functional classification of blocks becomes important in the determination of high-level conditional constructs and will be addressed in the next subsection.

In addition to identifying blocks, it was necessary to assign a "type" to each block that would be useful during the second phase of decompilation. Block type could be any value in the range 1 to 7 depending on the particular combination of the following three properties exhibited by a given block.

1) The block contained an entry point of the module.
2) The block contained module exit or return.
3) The next sequential block could not be reached except by direct transfer.

Table 1 shows the correspondence between block type and the presence of the attributes above. An "x" entry indicates the presence of an attribute.

TABLE 1. BLOCK TYPE

	ATTRIBUTES		
	1	2	3
0			
1	x		
2		x	
3	x	x	
4			x
5	x		x
6		x	x
7	x	x	x

Block type was used not only for flow-structure analysis, but also for determining whether or not the high-level objective language should support a multiple-entry-point feature in the definition of subprograms or procedures. The results of the decompilaton revealed the figures displayed in Table 2. This data was based on a total of 33 modules analyzed.

TABLE 2. ENTRY POINT DATA

Number of Entry Points	Frequency (No. of Modules).
1	22
2	3
> 2	8

Since 33 percent of the modules had more than one entry point, it was apparent the multiple-entry feature should be included in the high-level language.

INSTRUCTION CLASSIFICATION

The primary source of information used to identify primitive data types for the high-level languge (HLL) was the instruction class frequency distribution. By carefully classifying the instructions of the SIGMA-7, we hoped to identify the use of one or more of the following data types.

1) integer
2) single precision floating point
3) double precision floating point
4) string
5) logical (true/false)
6) Boolean (bit-string)
7) stacks.

We have included "stacks" in this list because the SIGMA-7 has specific instructions for manipulating stacks. Twenty-one instruction classes were selected in an attempt to identify the use of one or more of the data types above.

1) Bit Operations/Non-Compare
2) Bit Operation/Compare
3) Byte Operations/Non-Compare
4) Byte Operations/Compare
5) Half-Word Operations/Non-Compare
6) Half-Word Operations/Compare
7) Full-Word Operations/Non-Compare
8) Full-Word Operations/Compare

9) Double-Word Operations/Non-Compare
10) Double-Word Operations/Compare
11) Float-Short/Non-Compare
12) Float-Short/Compare
13) Float-Long/Non-Compare
14) Float-Long/Compare
15) Logical (OR,EOR,AND)
16) Stack Operations
17) Branch-On-Condition
18) Branch-On-Count
19) Branch (Subroutine Linkage)
20) Miscellaneous
21) Exchange-Word.

Some comments are in order concerning the interpretation of some of these classes. Class 1 consisted primarily of shift instructions and instructions that set hardware conditions. Class 3 consisted of byte-string operations. Instructions in Class 5 were interpreted as indexing operations for loop control or for counting purposes. Classes 7 and 9 represent instructions used primarily for true integer arithmetic. Classes 11-16 are self-explanatory. Instructions in Class 17 were assumed to be used for conditional logic and control of non-counting loops. Class 18 denotes instructions used primarily to control counting loops. Class 20 included instructions primarily pertaining to I/O handling. The last class, 21, consisted of the "exchange-word" instruction. This instruction interchanges or "swaps" two memory words. Forming a single class from this one instruction was done to determine if a swap function should be included as a system function in HLL. Table 3 shows the results of our analysis for the 33, F4-modules. Percentages were based on a total of 7,614 instructions.

TABLE 3. INSTRUCTION-CLASS FREQUENCY DISTRIBUTION

PERCENT*	CLASS	DESCRIPTION
49.3	7	Full-Word/Non-Compare
17.1	17	Branch-On-Condition
16.1	11	Float-Short/Non-Compare
5.5	19	Subroutine Linkage
4.9	8	Full-Word/Compare
2.5	1	Bit-Operation/Non-Compare
2.1	15	Logical
0.7	20	Misc. (I/O)
0.6	2	Bit Operation/Compare
0.5	18	Branch-On-Count
0.4	9	Double-Word/Non-Compare
0.2	21	Exchange-Word
0.0	All Other	

*All percentages are rounded to nearest .1 percent.

The rather high percentage of instructions in Class 7 can be explained by noting that all full-word "load" and "store" operations belong to this class. In addition, the absence of representatives from Classes 5 and 6 suggest that indexing and counting was done using full-word operations. Although a more definitive classification scheme could have disclosed the precise composition of Class 7, it is clear that full-word binary integers together with the standard arithmetic operations on integer data must be supported in HLL. This proposition is further supported by the presence of classes 9, 18 and 8.

Practically all computation was done in Class 11, suggesting HLL should definitely support single-precison real arithmetic. In addition, manual inspection of some of the modules indicated the need to support all trigonometric functions as a standard part of the language. A very surprising statistic was the total absence of classes 13 and 14.

Logical data was clearly indicated by the presence of Class 15 instructions. The surprisingly high frequency of Class 1 and 2 instructions suggest some facility should be provided in HLL for defining bit-string data and performing bit-string operations. Manual inspection of the code indicated that most bit instructions were used to set and test "switches." Since approximately half the modules performed bit operations, it was apparent that bit data should be an essential feature of HLL.

The results of instruction-class data suggest the following data facilities should be supported in HLL.

1) Fixed-point integer data
2) Single-precision floating-point data
3) Bit variables and bit strings
4) Logical data
5) Arithmetic operations for Classes 1) and 2)
6) An Exchange-Word primitive function
7) Trigonometric primitive functions.

ADDRESSING MODES

The SIGMA-7 supports a variety of addressing capabilities. Five different addressing modes were identified for the purposes of our analysis. They are:

1) Direct or Absolute Addressing
2) Indexing via registers
3) Indirect addressing
4) A combination of 2) and 3)
5) Immediate

In addition to gathering frequency counts for these five addressing modes within each instruction class, frequency counts were also maintained for "external" references in each instruction class. An "external" reference is a reference to a data item or instruction defined in as

physically distinct module. In fact, of the 33 modules in the F4 software, 2 modules were strictly data modules; that is, they contained no executable instructions and served to resolve many of the external references made in other modules.

The presence of indexing usually suggests the use of array constructs of one or more dimensions. Indirect addressing is typically used in addressing parameters passed to a subroutine, returning from a subroutine call and in accessing and modifying linked data structures like lists and trees. The presence of external references to data implies the need for "global" data structures as provided by block structured languages like ALGOL on Pl/1, or "common" data similar to that supported by FORTRAN. Table 4 summarizes the statistics accumulated for addressing modes.

TABLE 4. ADDRESSING MODE STATISTICS

	PERCENT OF INSTRUCTIONS (All Modules)
DIRECT	64.7%
INDEXING	5.1%
INDIRECT	1.3%
INDIRECT AND INDEXING	0.4%
IMMEDIATE	5.5%
EXTERNAL	23.1%

Practically all instances of indexing in the F4 were for the purpose of array addressing. Indirect addressing was used in three contexts: First, in effecting a return from subroutine call; second, in accessing parameters passed to a subroutine; and finally, indirect addressing was used to access external data via indirect references through other external variables (sometimes with post-indexing). The first two uses of indirect addressing represent standard linkage mechanisms for invoking and returning from subprograms. The third use suggests the addition of a new data type to the HLL which we shall call "locator" data. A locator variable would take as its value the location or address of some other data structure. To support the locator data type would require including a unary address-operator that when applied to a data structure or variable returns its address. Locator variables would allow the building of linked data structures quite easily in HLL.

The high percentage of references to external data indicate a clear need for the COMMON feature found in FORTRAN. Distinct data areas independent of any particular module serving as a data communication medium between two or more modules is to be desired over the parameter passing mechanism. This is particularly true if the number of data items communicated is relatively large. Extensive use was made of this method of data communication throughout the F4 system.

CONTROL FLOW ANALYSIS

The second phase of decompilation was designed to construct a flow graph of the blocks identified in the first phase. In addition, statistics were gathered on the frequency of instruction sequences. Instruction sequences that computed the value of some expression were identified within each block and a count of their frequency of occurrence was maintained at both the block level and module level. It was hoped that by accumulating instruction sequence data, we could identify simple functions that should be supported in HLL. Unfortunately, the only sequences that occurred with significant regularity were sequences containing at most two operators. The analysis did show that logical expressions involving AND, OR and EXCLUSIVE-OR represented 4.63 percent of all sequences identified. This was interpreted as sufficient evidence for support of these operators in HLL.

LOOP ANALYSIS

Perhaps the most interesting results were derived from the identification and classification of "loop" constructs. Our purpose was to classify all loops into 8 categories according to the number and placement of loop termination points. The results would show in addition to the type of loop constructs needed in HLL, the "natural" frequency of occurrance of single-entry, single-exit loops like those proposed by Dijkstra[3].

A "loop" was defined as a sequence of blocks, B_1, B_2,..., B_n, satisfying:

1) $B_1 = B_n$ and no B_i other than $B_1(B_n)$ occurs more than once.
2) Control can reach B_1 from some module entry point without passing through any block in the sequence.
3) Control flow can pass from B_i to B_{i+1}, $1 \leq i < n$.

The "head block" of a loop is always its entry block, B_1. The "tail" of a loop is always the block B_{n-1}. An "interior block" is any block other than the head or tail. An "exit" block is any block from which control can directly pass to a block not in the loop. Table 5 classifies loops according to whether: 1) the head is an exit, 2) an interior block is an exit, 3) the tail is an exit or some combination of these possibilities. Note that type 0 loops represent infinite loops.

3. Dahl, O.J., Dijkstra, E.W. and Hoare, C.A.R., Structured Programming, Academic Press, New York, 1972.

TABLE 5. LOOP CLASSIFICATION

	Attributes Present		
Type	1	2	3
0			
1			x
2		x	
3		x	x
4	x		
5	x		x
6	x	x	
7	x	x	x

Attributes:
1 = Head Exit
2 = Interior Exit
3 = Tail Exit

The results of the loop classification were very interesting particularly because of the surprisingly low frequency of occurrence of the DO-WHILE loops, Class 4, and the REPEAT-UNTIL Loops, Class 1. Table 6 gives a complete summary of the loop analysis data.

TABLE 6. LOOP CLASSIFICATION DATA

Module Name	No. of Blocks	No. of Loops	LOOP DISTRIBUTION BY CLASS							
			0	1	2	3	4	5	6	7
ACI	106	0	-	-	-	-	-	-	-	-
ACS	97	0	-	-	-	-	-	-	-	-
APL	41	15	-	-	-	-	-	3	2	10
CIV	174	0	-	-	-	-	-	-	-	-
CPM	141	0	-	-	-	-	-	-	-	-
CTL	29	0	-	-	-	-	-	-	-	-
DC	49	15	-	-	-	6	-	9	-	-
DEC	41	0	-	-	-	-	-	-	-	-
DR	226	106	-	1	32	12	-	9	34	18
DRED	24	6	-	-	-	1	-	2	1	2
EGI	30	24	-	-	-	-	-	-	-	24
EPE	146	1	-	-	-	-	-	-	1	-
ESE	48	0	-	-	-	-	-	-	-	-
FBP	64	0	-	-	-	-	-	-	-	-
FG	6	2	-	-	-	-	-	2	-	-
FGMV	38	12	-	-	-	-	-	12	-	-
INTG	9	2	-	1	-	-	-	1	-	-
IT	40	1	-	-	-	-	-	-	1	-
LCE	90	0	-	-	-	-	-	-	-	-
LDE	14	0	-	-	-	-	-	-	-	-
LDCE	23	0	-	-	-	-	-	-	-	-
LE	35	0	-	-	-	-	-	-	-	-
MCC	41	1	1	-	-	-	-	-	-	-
MCG	5	0	-	-	-	-	-	-	-	-
PIC	79	32	-	1	-	-	2	4	1	24
RNG	1	0	-	-	-	-	-	-	-	-
RTIO	9	2	-	-	-	-	-	2	-	-
SIN	26	2	-	-	-	-	-	2	-	-
SS	36	0	-	-	-	-	-	-	-	-
STAR	31	13	-	-	2	-	-	8	3	-
TA	32	4	-	-	-	-	-	4	-	-
TABL	1	0	-	-	-	-	-	-	-	-
Totals	238		1	3	34	19	2	59	42	78

Based on the percentage of Branch-On-Count instructions in Table 3, the number of counting loops was estimated to be 16 percent of the total number identified.

The loop data suggests HLL constructs like those illustrated below. The "WHEN" statement causes the inner-most loop containing it to terminate with control being passed to the first statement following the corresponding "LOOP" statement. Any number of WHEN statements would be allowed within the body of a loop. A "counting" option is allowed only

on the loop entry statement, DO, while both the WHILE and UNLESS options are permitted on the loop-end statement, LOOP. "V" denotes a counting variable. "A" denotes an arithmetic expression, while "L" denotes a logical expression. Optional clauses are indicated by square brackets.

```
DO  [ V = A1 TO A2  BY A3 ]
    [ WHILE (L1)          ]  ;
    [ UNLESS (L1)         ]

    ---------
    ---------
    ---------

    WHEN (L2) LEAVE;

    ---------
    ---------
    ---------

LOOP [ WHILE (L3)   ]
     [ UNLESS (L3)  ]  .
```

CONDITIONAL LOGIC

Analysis of the control-flow graphs produced by the decompiler showed that only 5 out of 1,732 blocks had more than 2 successor blocks; each of the 5 had 3 successors. Furthermore, 7 blocks consisted of a single conditional branch instruction implying very few blocks had more than 3 successors; the 7 single conditional-branch blocks were distributed among 6 modules. A more detailed examination of single-entry/single-exit subgraphs with no internal loops showed that practically all conditional logic could be represented by IF-THEN and IF-THEN-ELSE constructs with compound THEN and ELSE clauses. Another high-level construct that occurred with high frequency was the conditional GOTO.

SECTION III

CONCLUSIONS

We summarize our findings by enumerating those features characterizing a high-level language (HLL) suitable for developing applications software for aircraft trainer systems similar to the F4. Many of the features we identify are common to many conventional programming languages. These include integer data, floating point data and the usual set of arithmetic operators that apply to these data types. Logical data and the logical operations of "AND", "OR" and "NOT" were also found to be necessary features in HLL. Static array structures and COMMON data areas, like those found in FORTRAN, are also necessary elements of HLL. HLL should also support facilities for permitting different data structures to share the same storage area; a facility like the EQUIVALENCE concept in FORTRAN would be appropriate. Because of the relatively heavy use of tabular and simple data that were constant or "read only" data structures, some facility must be provided in HLL for initializing variables and arrays at compile or load time. Facilities for defining procedures and functions with multiple entry points were also found to be desirable ingredients of a language designed for trainer systems. Although the GOTO construct must be included in HLL, its use within the F4 software was very seldom found to be the result of "irresponsible style" on the part of the programmer.

For the most part, the features we have listed above are common to a number of popular high-level programming languages. We complete our summary of the properties ascribed to HLL by discussing in a little more detail those features that were very prominent in the F4 software, but were unusual in some respect and therefore of special interest as a result of this research.

DATA STRUCTURES

Add LOCATOR and BIT(n) as new data types, where "n" denotes the length of a bit-string variable. LOCATOR variables take on values that represent the location or address of some other data item. LOCATOR variables can be typed, dimensioned, equivalenced, placed in COMMON or passed as parameters just as any other variables. BIT variables can be equivalenced to other variables.

DATA OPERATIONS

While both LOCATOR and BIT variables should be allowed in mixed mode expressions (where they should be treated as INTEGER data), distinct operations should be provided for each type. An ADDR built-in or system function should be available returning the location of its argument. Operations like .AND., .NOT., .OR. and .EOR. should be available for BIT variables. A system function should also be provided for exchanging the values of two variables.

CONTROL STRUCTURES

A loop-construct like the one illustrated below consists of a "DO" statement having an optional clause with three distinct forms together with a corresponding "LOOP" statement also having an option with two alternative forms. Within the loop body, at the same nesting level, can be any number of "WHEN" statements. Form 1 of the DO statement is for creating a counting loop with "V" denoting a control variable and A_1, A_2, and A_3 denoting arithmetic expressions. In Forms 2 and 3, "WHILE" takes a logical expression defining a condition for entering the loop body, whereas "UNLESS" takes an expression defining a condition for terminating the loop. The WHEN statement defines a condition for loop termination and causes control to pass to the statement immediately following the LOOP statement if the condition is met.

```
DO [V = A1, A2, [A3]]      ;
   [WHILE (E)]
   [UNLESS (E)]

WHEN (E) LEAVE

LOOP[WHILE (E)]            ;
    [UNLESS (E)]
```

In addition to a new loop construct, it would be desirable to have conditional constructs like IF-THEN or IF-THEN-ELSE with compound clauses. The IF-THEN statement would introduce a group of statements to be executed only if the logical expression "E" is "true." The statement group would be terminated by a matching "ENDIF" or "ELSE" statement. An occurrence of the "ELSE" would signal the beginning of another statement group to be executed only if "E" is "false." The ELSE group would be terminated by the ENDIF. The IF-THEN and IF-THEN-ELSE constructs could be nested to any depth.

```
IF (E) THEN
_________
_________      THEN-group
_________

ELSE

_________
_________
_________      ELSE-group
_________
_________

ENDIF
```

INPUT/OUTPUT

While some evidence existed within the decompilation data suggesting facilities should be provided for handling I/O interrupts, it is not clear that input/output handling at this level should be supported in HLL; this function is most appropriately performed within the operating system. A compromise would be to allow HLL system functions that could check for the occurrence of various kinds of interrupt conditions, returning control only if the condition occurred or had not occurred.

SUMMARY

Real-time programs must be efficient in their use of processor time. Programming at the machine level is one way to guarantee a certain level of performance. Programming real-time applications in a high-level language has obvious advantages, but if performance of the object code is of paramount importance, the choice of language or language features probably has only secondary effects. Using good programming techniques and employing a good optimizing compiler will most probably have the greatest impact on performance.

While this project has sought to identify those features essential to a high-level language suitable and "natural" as possible for implementing real-time trainer systems, it has not determined the degree to which language design effects the ultimate performance of such systems. The question of design effectiveness remains open and represents an area of further research.

SECTION IV

RECOMMENDATIONS

The decompilation approach to designing a high-level language for trainer applications has been fruitful and has produced some unexpected results. Nevertheless, it is difficult to evalute the success of this approach after having applied it to only one system, the F4. Further studies are needed employing this technique with other types of trainer programs. In addition, a formal specification of HLL should be developed along with a compiler using a variety of optimization algorithms. Only by comparing the performance of compiled, unoptimized and optimized HLL code with the original handwritten machine code can we determine the most significant factors in writing and maintaining high-performance trainer application systems.

BIBLIOGRAPHY

1. Allen, F.E., "Control low analysis," ACM SIGPLAN Notices V, July, 1970, pp. 1-19.

2. Brown, P.J., "Levels of Language for Portable Software," CACM (15, 12), December, 1972, pp. 1059-1062.

3. Dellert, G.T., 1965. "A Use of Macros in Translation of Symbolic Assembly Language of One Computer to Another," CACM, Vol. 8, No. 12 (December, 1965), pp. 742-748.

4. Fisher, D.A., "A Survey of Control Structures in Programming Languages," ACM SIGPLAN Notices (7,11), November, 1972, pp. 1-14.

5. Gaines, R.S., 1965. "On the Translation of Machine Language Programs," CACM, Vol. 8, No. 12 (December, 1965), pp. 736-741.

6. Graham, M.L., Ingerman, P.Z., 1965. "An Assembly Language for Reprogramming," CACM, Vol. 8, No. 12 (December, 1965), pp. 769-773.

7. Gunn, J.H., 1962. "Problems in Program Interchangeability," Symbolic Languages in Data Processing. *New York: Gordon and Beach Science* Publishers, 1962, pp. 777-790.

8. Halstead, M.H., "Using Computers for Program Conversion," Datamation, May, 1970, pp. 125-129.

9. Hollander, Clifford R., Decompilation of Object Programs. Digital Systems Laboratory, Technical Report No. 54, Stanford University, January, 1973.

10. Housel, Barron C., A Study of Decompiling Machine Languages Into High-Level Machine Independent Languages,: Ph.D. Thesis, Department of Computer Scences, Purdue University, August, 1973.

11. Housel, Barron C., and Maurice H. Halstead, "A Methodology for Machine Language Decompilation," IBM Research Report RJ 1316 (No. 20557), San Jose, California, December, 1973.

12. Opler Ascher, et. al., 1962. "Automotic Translation of Programs From One Computer to Another," Proceedings IFIP Congress, 1962, pp. 550-553.

13. Sassaman, W.A., "A Computer Program to Translate Machine Language to FORTRAN," Proceedings SJCC, 1966, pp. 235-241.

14. Schneider, V.B., and Gary Winiger, "Translation Grammars for Compilation and Decompilation,: BIT, Volume 14, 1974, pp. 78-86.

GLOSSARY

addressing mode: the process by which a memory operand is located or fetched for a given machine instruction.

bit-string: a sequence of binary digits treated as a single unit or operand in some operation.

block: a sequence of machine instructions whose only entry point is the first instruction and for which any transfer instructions occur at the end of the sequence.

branch-on-condition: a type of conditional transfer or "jump" instruction that is based on the setting of hardware condition flags preset by the execution of an earlier instruction.

branch-on-count: a type of conditional transfer or "jump" instruction based on the value of some counter normally held in a register.

compiler: a process or processor designed to translate procedures written in a source language to equivalent procedures expressed in machine or assembly language.

control structure: a programming language statement or group of statements designed to determine the flow or sequence of executable expressions.

decompilation: the inverse process of compilation; that is, translating machine or assembly language procedures into a language that is more programmer oriented.

entry point: the location to which control is passed to execute or activate a program module or block within a module.

exit block: a block within a loop from which control can leave the loop.

external reference: a reference, either fetching an operand or transferring control, generated in one module but resolved to or satisfied by another module.

flow graph: a collection of nodes connected by directed arcs. Nodes correspond to program blocks. The arcs define the blocks immediately accessible from a given block.

global data: data that is accessible in one module but not defined within that module.

head block: the block containing the entry point to a loop.

immediate addressing: the method of fetching an instruction operand whose value forms part of the instruction being executed.

indexing: the use of registers in adjusting or computing an addressable location in memory that varies during the execution of a program segment.

indirect reference: a reference to a data item or instruction operand obtained by fetching the value of another memory location holding the reference address.

interior block: any block within a loop other than the head or tail blocks.

loop: a sequence of instructions or blocks that can be repeatedly executed as a result of the control flow structure of a module or program.

module: a group of program statements performing a specific function within the logical organization of a program.

parameter passing: the mechanism or processing convention used to transmit a set of data items from one module to another where the specific set of items may vary from one call of that module to another.

source code: a program written in a given language serving as the input to be translated by a compiler or decompiler.

subroutine linkage: the instruction sequence executed to transfer control from one module to another.

syntax: a finite set of rules describing how to construct programs expressed in a given language.

tail block: the block last executed before the head block within a loop.